New Features of ANSI C++ Standard

16

Key Concepts

Boolean type data | Wide-character literals | Constant casting | static casting | Dynamic casting | Reinterpret casting | Runtime type information | Explicit constructors | Mutable member data | Namespaces | Nesting of namespaces | Operator keywords | Using new keywords | New style for headers

16.1 Introduction

The ISO / ANSI C++ Standard adds several new features to the original C++ specifications. Some are added to provide better control in certain situations and others are added for providing conveniences to C++ programmers. It is therefore important to note that it is technically possible to write full-fledged programs without using any of the new features. Important features added are:

1. New data types
   - bool
   - wchar_t

2 New operators
   - const_cast
   - static_cast
   - dynamic_cast
   - reinterpret_cast
   - typeid

3. Class implementation
   - Explicit constructors
   - Mutable members

4. Namespace scope
5. Operator keywords
6. New keywords

7. New headers

We present here a brief overview of these features.

16.2 New Data Types

The ANSI C++ has added two new data types to enhance the range of data types available in C++. They are **bool** and **wchar_t.**

The bool Data Type

The data type **bool** has been added to hold a Boolean value, **true** or **false.** The values **true** and **false** have been added as keywords to the C++ language. The **bool** type variables can be declared as follows.

```
bool b1;              // declare bl as bool type
b1 = true;            // assign true value to it
bool b2 = false;      // declare and initialize
```

The default numeric value of **true** is 1 and **false** is 0. Therefore, the statements

```
cout << "bl = " << bl;       // b1 is true
cout << "b2 = " << b2;       // b2 is false
```

will display the following output:

```
b1 = 1
b2 = 0
```

We can use the **bool** type variables or the values **true** and **false** in mathematical expressions. For instance,

```
int x = false + 5*m - b1;
```

is valid and the expression on the right evaluates to 9 assuming **bl** is true and **m** is 2. Values of type **bool** are automatically elevated to integers when used in nonBoolean expressions.

It is possible to convert implicitly the data types pointers, integers or floating point values to **bool** type. For example, the statements

```
bool x = 0;
bool y = l00;
bool z = 15.75;
```

assign **false** to x and **true** to **y** and **z**.

Program 16.1 demonstrates the features of **bool** type data.

Program 16.1 Use of bool Type Data

```
#include<iostream>
using namespace std;

int main()

{
int arr1[5] = {1, 99, 3, -1, 6};
int arr2[5] = {0, 66, 3, 0, 6};
bool comp[5];

cout<<"Array 1: ";
for(int i=0; i<5;i++)
cout<<arr1[i]<<"\t";

cout<<"\nArray 2: ";
for(i=0; i<5;i++)
cout<<arr2[i]<<"\t";
```

```
cout<<"\nComparing the two arrays ..\n";
for(i=0; i<5;i++)
comp[i]=arr1[i]==arr2[i];

cout<<"\nDisplaying the comparison results..\n";
for(i=0; i<5;i++)
if(comp[i]==1)
      cout<<"Index "<<i<<" - Match\n";
else
      cout<<"Index "<<i<<" - No Match\n";

return 0;
}
```

The output of program 16.1 would be:

```
Array 1:1 99 3 -1 6
Array 2:0 66 3 0 6

Comparing the two arrays ..

Displaying the comparison results..
Index 1 - No Match
Index 2 - No Match
Index 3 - Match
Index 4 - No Match
Index 5 - Match
```

The wchar_t Data Type

The character type **wchar_t** has been defined in ANSI C++ to hold 16-bit wide characters. The 16-bit characters are used to represent the character sets of languages that have more than 255 characters, such as Japanese. This is important if we are writing programs for international distribution.

ANSI C++ also introduces a new character literal known as *wide_character* literal which uses two bytes of memory. Wide_character literals begin with the letter L, as follows:

```
L'xy'          // wide_character literal
```

16.3 New Operators

We have used cast operators (also known as *casts* or *type casts)* earlier in a number of programs. As we know, casts are used to convert a value from one type to another. This is necessary in situations where automatic conversions are not possible. We have used the following forms of casting:

```
double x = double(m);          // C++ type casting
double y = (double)n;          // C-type casting
```

Although these casts still work, ANSI C++ has added several new cast operators known as *static casts, dynamic casts, reinterpret casts* and *constant casts.* It also adds another operator known as **typeid** to verify the types of unknown objects.

The static_cast Operator

Like the conventional cast operators, the **static_cast** operator is used for any standard conversion of data types. It can also be used to cast a base class pointer into a derived class pointer. Its general form is:

```
static_cast<type>(object)
```

Here, *type* specifies the target type of the cast, and *object* is the object being cast into the new type. Examples:

```
int m = 10;
double x = static_cast<double> (m);
```

```
char ch = static_cast<char> (m);
```

The first statement casts the variable **m** to type **double** and the second casts it to type **char.**

Why use this new type when the old style still works? The syntax of the old one blends into the rest of the lines and therefore it is difficult to locate them. The new format is easy to spot and to search for using automated tools.

The const_cast Operator

The **const_cast** operator is used to explicitly override **const** or **volatile** in a cast. It takes the form

const_cast<*type*>(*object)*

Since the purpose of this operator is to change its **const** or **volatile** attributes, the *target type* must be the same as the *source type.* It is mostly used for removing the const_ness of an object.

The reinterpret_cast Operator

The **reinterpret_cast** is used to change one type into a fundamentally different type. For example, it can be used to change a pointer type object to integer type object or vice versa. It takes the following form:

reinterpret_cast<*type*>(*object)*

This operator should be used for casting inherently incompatible types. Examples:

```
int m;
float x;
```

```
int* intptr;
float* floatptr;
intptr = reinterpret_cast<int*> (m);
floatptr = reinterpret_cast<float*> (x);
```

The dynamic_cast Operator

The dynamic cast is used to cast the type of an object at runtime. Its main application is to perform casts on polymorphic objects. Recall that polymorphic objects are created by base classes that contain virtual functions. It takes the form:

dynamic_cast<*type*>(*object)*

The *object* is a base class object whose type is to be checked and casted. It casts the type of object to *type.* It never performs an invalid conversion. It checks that the conversion is legal at runtime. If the conversion is not valid, it returns NULL.

The *type* must be a pointer or a reference to a defined class type. The *argument* object must be expression that resolves to a pointer or reference. The use of the operator **dynamic_cast()** is also called a *type-safe downcast.*

The typeid Operator

We can use the **typeid** operator to obtain the types of unknown objects, such as their class name at runtime. For example, the statement

char *objectType = **typeid**(*object).* **name();**

will assign the type of “object” to the character array **objectType** which can be printed out, if necessary. To do this, it uses the **name()**

member function of the **type_info** class. The object may be of type **int, float,** etc. or of any class.

We must include **<typeinfo>** header file to use the operators **dynamic_cast** and **typeid** which provide run-time type information (RTTI).

As the name suggests, RTTI is the process of determining the dynamic type of an object at runtime. While we may have declared a pointer with some static type, it is quite possible that at run time it may point to any class type derived from the static type. The following example demonstrates RTTI using typeid operator:

```
#include <iostream>
#include <typeinfo>
class Org
{

        public:
            // Members of Org class

        virtual ~Org() {}
};

class Dept : public Org
{
            // Members of Dept class
};

int main ()
{
        Org org1;
        Dept dept1;
        Org *ptr = &dept1;

        std::cout << typeid(ptr).name() << std::endl;
        // This will display Org * i.e. the static pointer type
```

```
        std::cout << typeid(*ptr).name() << std::endl;
        // This will display Dept i.e. the dynamic pointer type

}
```

In the above program, the typeid (object) call returns a reference to the type_info object that contains information about the object's type. The member function name () of the type_info class is used for retrieving the name of the object type.

16.4 Class Implementation

ANSI C++ Standard adds two unusual keywords, **explicit** and **mutable,** for use with class members.

The explicit Keyword

The **explicit** keyword is used to declare class constructors to be "explicit" constructors. We have seen earlier, while discussing constructors, that any constructor called with one argument performs *implicit conversion* in which the type received by the constructor is converted to an object of the class in which the constructor is defined. Since the conversion is automatic, we need not apply any casting. In case, we do not want such automatic conversion to take place, we may do so by declaring the one-argument constructor as explicit as shown below:

```
class ABC
{
        int m;
    public:
        explicit ABC (int i)            // constructor
        {
            m = i;
        }
        // .....
```

```
        // .....
};
```

Here, objects of ABC class can be created using only the following form:

```
ABC abc1(100);
```

The automatic conversion form

```
ABC abcl = 100;
```

is not allowed and is also illegal. Remember, this form is permitted when the keyword **explicit** is not applied to the conversion.

The mutable Keyword

We know that a class object or a member function may be declared as **const** thus making their member data not modifiable. However, a situation may arise where we want to create a **const** object (or function) but we would like to modify a particular data item only. In such situations we can make that particular data item modifiable by declaring the item as **mutable.** Example:

```
mutable int m;
```

Although a function(or class) that contains **m** is declared **const,** the value of **m** may be modified. Program 16.2 demonstrates the use of a **mutable** member.

Program 16.2 Use of Keyword mutable

```
#include <iostream>
using namespace std;
```

```
class ABC
{
        private:
            mutable int m;              // mutable member
        public:
            explicit ABC(int x = 0)
            {
            m = x;
            }
            void change() const             // const function
            {
                        m = m+10;
            }
            int display() const             // const function
            {
                        return m;
            }
};
int main()
{
            const ABC abc(100);             // const object
            cout << "abc contains: " << abc.display();
            abc.change();           // changes mutable data

            cout << "\nabc now contains: " << abc.display();
            cout << "\n";

            return 0;
}
```

The output of Program 16.2 would be:

```
abc contains: 100
abc now contains: 110
```

NOTE: *Although the function **change()** has been declared constant, the value of **m** has been modified. Try to execute the program after deleting the keyword **mutable** in the program.*

16.5 Namespace Scope

We have been defining variables in different scopes in C++ programs, such as classes, functions, blocks, etc. ANSI C++ Standard has added a new keyword **namespace** to define a scope that could hold global identifiers. The best example of namespace scope is the C++ Standard Library. All classes, functions and templates are declared within the namespace named **std.** That is why we have been using the directive

```
using namespace std;
```

in our programs that uses the standard library. The **using namespace** statement specifies that the members defined in **std** namespace will be used frequently throughout the program.

Defining a Namespace

We can define our own namespaces in our programs. The syntax for defining a namespace is similar to the syntax for defining a class. The general form of namespace is:

```
namespace  namespace_name
{
        // Declaration of
        // variables, functions, classes, etc.
}
```

NOTE: *There is one difference between a class definition and a namespace definition. The namespace is concluded with a closing brace but no terminating semicolon.*

Example:

```
namespace TestSpace
{
        int m;
        void display(int n)

        {
            cout << n;
        }
}                           // No semicolon here
```

Here, the variable **m** and the function **display** are inside the scope defined by the **TestSpace** namespace. If we want to assign a value to **m,** we must use the scope resolution operator as shown below.

```
TestSpace::m = 100;
```

Note that **m** is qualified using the namespace name.

This approach becomes cumbersome if the members of a namespace are frequently used. In such cases, we can use a **using** directive to simplify their access. This can be done in two ways:

```
using namespace namespace_name;          // using directive

using namespace_name::member_name;       // using
declaration
```

In the first form, all the members declared within the specified namespace may be accessed without using qualification. In the second form, we can access only the specified member in the program.

Example:

```
using namespace TestSpace;
m = 100;            // OK
display(200);       //OK

using TestSpace::m;
m = 100;            // OK
display(200);       // Not ok, display not visible
```

Nesting of Namespaces

A namespace can be nested within another namespace. Example:

```
namespace NS1
{
        .....
        .....
        namespace NS2
        {
            int m = 100;
        }
        .....
        .....
}
```

To access the variable **m,** we must qualify the variable with both the enclosing namespace names. The statement

```
cout << NS1::NS2::m;
```

will display the value of **m.** Alternatively, we can write

```
using namespace NS1;
cout << NS2::m;
```

Unnamed Namespaces

An unnamed namespace is one that does not have a name. Unnamed namespace members occupy global scope and are accessible in all scopes following the declaration in the file. We can access them without using any qualification.

A common use of unnamed namespaces is to shield global data from potential name classes between files. Every file has its own, unique unnamed namespace.

Program 16.3 demonstrates how namespaces are defined, how they are nested and how an unnamed namespace is created. It also illustrates how the members in various namespaces are accessed.

Program 16.3 Using Namespace Scope with Nesting

```
#include <iostream>

using namespace std;

// Defining a namespace
namespace Name1
{
    double x = 4.56;
    int m = 100;

    namespace Name2          // Nesting namespaces
    {
```

```
        double y = 1.23;
        }
}
namespace                       // Unnamed namespace
{
        int m = 200;
}

int main()
{

        cout << "x = " << Name1::x << "\n"; // x is qualified
        cout << "m = " << Name1::m << "\n";
        cout << "y = " << Name1::Name2::y << "\n"; // y is fully
                                                   // qualified
        cout << "m = " << m << "\n";       // m is global

        return 0;
}
```

The output of Program 16.3 would be:

```
x = 4.56
m = 100
y = 1.23
m = 200
```

NOTE: *We have used the variable **m** in two different scopes.*

Program 16.4 shows the application of both the **using** directive and **using** declaration.

Program 16.4 Illustrating the using Keyword

```
#include <iostream>

using namespace std;

// Defining a namespace
namespace Name1
{
        double x = 4.56;
        int m = 100;

        namespace Name2                    // Nesting namespaces
        {
            double y = 1.23;
        }
}
namespace Name3
{
        int m = 200;
        int n = 300;
}
int main()
{
        using namespace Name1;              // bring members of
          Name1
                                    // to current scope
        cout << "x = " << x << "\n";        // x is not qualified
        cout << "m = " << m << "\n";
        cout << "y = " << Name2::y << "\n";    // y is qualified
        using Name3::n;                // bring n to current scope
        cout << "m = " << Name3::m <<"\n";     // m is qualified
        cout << "n = " << n << "\n";          // n is not qualified
        return 0;
}
```

The output of Program 16.4 would be:

```
x = 4.56
m = 100
y = 1.23
m = 200
n = 300
```

NOTE: *Understand how the data members are qualified when they are accessed.*

Program 16.5 uses functions in a namespace scope.

Program 16.5 Using Functions in Namespace Scope

```
#include <iostream>

using namespace std;

namespace Functions
{
     int divide(int x,int y)          // definition
     {
          return(x/y);
     }
     int prod(int x,int y);           // declaration only
}

int Functions:: prod(int x,int y)       // qualified
{
     return(x*y);
}
int main()
```

```
{
    using namespace Functions;

    cout << "Division: " << divide(20,10) << "\n";
    cout << "Multiplication: " << prod(20,10) << "\n";

    return 0;
}
```

The output of Program 16.5 would be:

```
Division: 2
Multiplication: 200
```

NOTE: *When a function that is declared inside a namespace is defined outside, it should be qualified.*

Program 16.6 demonstrates the use of classes inside a namespace.

Program 16.6 Using Classes in Namespace Scope

```
include <iostream>
using namespace std;
namespace Classes
{
        class Test
{
    private:
         int m;
```

```
    public:
        Test(int a)
        {
            m = a;
        }
        void display()
        {
                cout << "m = " << m << "\n";
        }
}
};
int main()
{
        // using scope resolution
        Classes:: Test T1(200);
        T1.display();

        // using directive
        using namespace Classes;
        Test T2(400);
        T2.display();

        return 0;
}
```

The output of Program 16.6 would be:

```
m = 200
m = 400
```

16.6 Operator Keywords

The ANSI C++ Standard proposes keywords for several C++ operators. These keywords, listed in Table 16.1, can be used in place of operator symbols in expressions. For example, the expression

```
x > y && m != 100
```

may be written as

```
x > y and m not_eq 100
```

Operator keywords not only enhance the readability of logical expressions but are also useful in situations where keyboards do not support certain special characters such as &, ^ and ~.

Table 16.1 *Operator keywords*

Operator	*Operator keyword*	*Description*
&&	and	logical AND
||	or	logical OR
!	not	logical NOT
!=	not_eq	inequality
&	bitand	bitwise AND
|	bitor	bitwise inclusive OR
^	xor	bitwise exclusive OR
~	compl	bitwise complement
&=	and_eq	bitwise AND assignment
|=	or_eq	bitwise inclusive OR assignment
^=	xor_eq	bitwise exclusive OR assignment

16.7 New Keywords

ANSI C++ has added several new keywords to support the new features. Now, C++ contains 64 keywords, including **main.** They are listed in Table 16.2. The new keywords are boldfaced.

Table 16.2 *ANSI C++ keywords*

asm	*else*	*namespace*	*template*
auto	enum	new	this
bool	**explicit**	operator	throw
break	**export**	private	**true**
case	extern	protected	try
catch	**false**	public	typedef
char	float	register	**typeid**
class	for	**reinterpret_cast**	**typename**
const	friend	return	union
const_cast	goto	short	unsigned
continue	if	signed	**using**
default delete	inline int	sizeof static	virtual void
do	long	**static-cast**	volatile
double	main	struct	**wchar_t**
dynamic_cast	**mutable**	switch	while

16.8 New Headers

The ANSI C++ Standard has defined a new way to specify header files. They do not use **.h** extension to filenames. Example:

```
#include <iostream>
#include <fstream>
```

However, the traditional style **<iostream.h>, <fstream.h>,** etc., is still fully supported. Some old header files are renamed as shown below:

Old style	*New style*
<assert.h>	<cassert>
<ctype.h>	<cctype>
<float.h>	<cfloat>

<limits.h>	<climits>
<math.h>	<cmath>
<stdio.h>	<cstdio>
<stdlib.h>	<cstdlib>
<string.h>	<cstring>
<time.h>	<ctime>

Summary

- ANSI C++ Standard committee has added many new features to the original C++ language specifications.
- Two new data types **bool** and **wchar_t** have been added to enhance the range of data types available in C++.
- The **bool** type can hold Boolean values, **true** and **false.**
- The **wchar_t** type is meant to hold 16-bit character literals.
- Four new cast operators have been added: static_cast, const_cast, reinterpret_cast and dynamic_cast.
- The **static_cast** operator is used for any standard conversion of data types.
- The **const_cast** operator may be used to explicitly change the **const** or **volatile** attributes of objects.
- We can change the data type of an object into a fundamentally different type using the **reinterpret_cast** operator.
- Casting of an object at run time can be achieved by the **dynamic_cast** operator.
- Another new operator known as **typeid** can provide us run time type information about objects.

- A constructor may be declared **explicit** to make the conversion explicit.
- We can make a data item of a **const** object or function modifiable by declaring it **mutable**.
- ANSI C++ permits us to define our own **namespace** scope in our program to overcome certain name conflict situations.
- Namespaces may be nested.
- Members of **namespace** scope may be accessed using either **using** declaration or **using** directive.
- ANSI C++ proposes keywords that may be used in place of symbols for certain operators.
- In new standard, the header files should be specified without **.h** extension and the **using** directive

  using namespace std;

  should be added in every program.
- Some old style header files are renamed in the new standard. For example, **math.h** file is known as **cmath.**

Key Terms

and | and_eq | ANSI C++ **| bitand | bitor | bool |** Boolean **|** C_type casting **|** C++ standard **|** C++ type casting **|** casts **| compl | const | const** function **| const** object **| const**_cast **|** constant casts **|** current scope **|** downcast **| dynamic_casts |** dynamic casts **| explicit** constructor **| false** value **|** global identifiers **| header file |** implicit conversion **| mutable** member **| name()** function **| namespace** scope **|** nesting namespaces **| not | not_eq |** operator keywords **| or | or_eq |** polymorphic objects **|** reinterpret casts **| reinterpret_cast |** RTTI **|** source type **|** standard library **|** static

casts | **static_cast** | **std** namespace | target type | **true** value | type casts | **type_info** class | type_safe casting | **typeid** | **typeinfo** header | unnamed namespaces | **using** declaration | **using** directive | **using namespace** | **volatile** | **wchar_t** | wide_character literal | **xor** | **xor_eq**

Review Questions

16.1 List the two data types added by the ANSI C++ standard committee.

16.2 What is the application of **bool** type variables?

16.3 What is the need for **wchar_t** character type?

16.4 List the new operators added by the ANSI C++ standard committee.

16.5 What is the application of **const_cast** operator?

16.6 Why do we need the operator **static_cast** while the old style cast does the same job?

16.7 How does the **reinterpret_cast** differ from the **static_cast?**

16.8 What is dynamic casting? How is it achieved in C++?

16.9 What is **typeid** operator? When is it used?

16.10 What is explicit conversion? How is it achieved?

16.11 When do we use the keyword **mutable?**

16.12 What is a namespace conflict? How is it handled in C++?

16.13 How do we access the variables declared in a named namespace?

16.14 What is the difference between using the **using namespace** directive and using the **using** declaration for accessing **namespace** members?

16.15 What is wrong with the following code segment?

```
const int m = 100;
int *ptr = &m;
```

16.16 What is the problem with the following statements?

```
const int m = 100;
double *ptr = const_cast<double*>(&m);
```

16.17 What will be the output of the following program?

```
#include<iostream.h>
class Person
{
      // .....
}
int main()
{
      Person John;
      cout << " John is a ";
      cout << typeid(John).name() << "\n";
}
```

16.18 What is wrong with the following namespace definition?

```
namespace Main
{
      int main()
      {
         // .....
      }
}
```

Debugging Exercises

16.1 Identify the error in the following program.

```
#include <iostream>
class A
{
public:
        A()
        {
        }
        A(int i)
        {
        }
};
class B
{
public:

        B()
        {
        }
        explicit B(int)

        {
        }
};
void main()
{
        A a1=12;
        A a2;
        A a3=a1;
        B b1 = 12;
}
```

16.2 Identify the error in the following program.

```
#include <iostream.h>

class A
{
protected:
```

```
        int i;

public:
        A()
        {
            i = 10;
        }
        int getI()
        {

            return i;
        }
};
class B: public A
{
public:
        B()
        {
        }
        int getI()
        {
            return i + i;
        }
};
void main()
{
        A *a = new A();
        B *b = static_cast<B*>(a);
        cout << b->getI();
}
```

16.3 Identify the error in the following program.

```
#include <iostream.h>

namespace A
{
        int i;
```

```
        void displ()
        {
            cout << i;
        }
}
void main()
{
        namespace Inside
        {
            int insideI;
            void displnsideI()
            {
                cout << insideI;
            }
        }
        A::i = 10;
        cout << A::i;
        A::displ();

        Inside::insideI = 20;
        cout << Inside::insideI;
        Inside::displnsideI();
}
```

Programming Exercises

16.1 Write a program to demonstrate the use of **reinterpret_cast** operator  .

16.2 Define a namespace named **Constants** that contains declarations of some constants. Write a program that uses the constants defined in the namespace **Constants**  .